Solving String Constraints Lazily *

Pieter Hooimeijer and Westley Weimer
University of Virginia
{pieter, weimer}@cs.virginia.edu

ABSTRACT

Decision procedures have long been a fixture in program analysis, and reasoning about string constraints is a key element in many program analyses and testing frameworks. Recent work on string analysis has focused on providing decision procedures that model string operations. Separating string analysis from its client applications has important and familiar benefits: it enables the independent improvement of string analysis tools and it saves client effort.

We present a constraint solving algorithm for equations over string variables. We focus on scalability with regard to the size of the input constraints. Our algorithm performs an explicit search for a satisfying assignment; the search space is constructed lazily based on an automata representation of the constraints. We evaluate our approach by comparing its performance with that of existing string decision procedures. Our prototype is, on average, several orders of magnitude faster than the fastest existing implementation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; F.3.1 [**Logics and Meanings**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

General Terms

Algorithms, Languages, Theory, Verification

Keywords

regular language, decision procedure, scalability

*This research was supported in part by National Science Foundation Grants CCF 0954024, CCF 0916872, CNS 0716478, CNS 0627523 and Air Force Office of Scientific Research grant FA9550-07-1-0532, as well as gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

1. INTRODUCTION

Reasoning about string variables is a key aspect in many areas of program analysis [2, 20, 26, 29, 32] and automated testing [7, 8, 9, 19]. Program analyses and transformations that deal with string-manipulating programs, such as test input generation for legacy systems [15, 16], web application bug finding [29], and program repair [31], invariably require a model of string manipulating functions.

Traditionally, both static and dynamic analyses have relied on their own built-in models to reason about constraints on string variables, just as early analyses relied on built-in conservative reasoning about aliasing. The current situation is suboptimal for two reasons: first, it forces researchers to reinvent the wheel for each new tool; and second, it inhibits the independent improvement of algorithms for reasoning about strings.

External constraint solving tools have long been available for other domains, such as satisfiability modulo theories (SMT) [3, 4, 23] and boolean satisfiability (SAT) [5, 21, 33]. Recent work in string analysis has focused on providing similar external decision procedures for string constraints [1, 10, 12, 27, 28, 34]. Thus far, this work has focused on adding features such as support for symbolic integer constraints [34], support for bounded context-free grammars [1, 12], and embedding into an existing SMT solver [3, 27]. We argue that the existing approaches leave significant room for improvement with regard to scalability.

We propose a novel decision procedure that supports the efficient, lazy processing of string constraints without requiring a priori length bounds. Our approach is based on the insight that existing solvers do more work than is strictly necessary because they eagerly encode the search space of possible solutions before searching it. For example, the Hampi tool [12] performs an eager bitvector encoding of all positional shifts for each regular expression in the given constraint system. We observe that much of that encoding work is unnecessary if the goal is to find a single string assignment as quickly as possible.

Our approach uses an automaton-based representation of string constraint systems. In contrast with previous automaton-based approaches [10, 12, 27, 28, 34], we separate the description of the search space from its instantiation. For example, when intersecting two automata using the cross-product construction [25], we generate only those parts of the intersection automaton needed to find a single string. Our search space consists of sets of nodes in lazily-constructed finite automata corresponding to string variables and constrained by string operations.

Constraint	::=	$StringExpr \in RegExpr$	inclusion
	|	$StringExpr \notin RegExpr$	non-inclusion
StringExpr	::=	*Var*	string variable
	|	$StringExpr \circ Var$	concat
RegExpr	::=	*ConstVal*	string literal
	|	$RegExpr + RegExpr$	language union
	|	$RegExpr\ RegExpr$	language concat
	|	$RegExpr^\star$	Kleene star

Figure 1: String inclusion constraints for regular sets. A constraint system is a set of constraints over a shared set of string variables; a satisfying assignment maps each string variable to a value so that all constraints are simultaneously satisfied. *ConstVal* represents a string literal; *Var* represents an element in a finite set of shared string variables.

```
1: follow_graph(I : constraint system) =
2:   let G : directed graph = empty
3:   let M : constraint → path = empty
4:   foreach C_i : constraint ∈ I do
5:     let (v_1 ∘ ... ∘ v_n ◇ R) = C_i
6:     for j ∈ 1, ..., n − 1 do
7:       G ← add_edge(G, node(v_j), node(v_{j+1}))
8:     M[C_i] ← [node(v_1), ..., node(v_n)]
9:   return (G, M)
```

Figure 2: Follow graph generation. Given a constraint system I, we output follow graph G and mapping M (defined in the text). G and M capture the high-level structure of the search space of assignments. The node **function returns a distinct vertex for each variable.**

The primary contributions of this paper are:

- A novel decision procedure that supports the efficient and lazy analysis of string constraints. We treat string constraint solving as an explicit search problem, and separate the description of the search space from the search strategy used to traverse it.
- A comprehensive performance comparison between our prototype tool and existing implementations. We compare against CFG Analyzer [1], DPRLE [10], and Hampi [12]. We use several sets of established benchmarks [12, 27, 28]. We find that our prototype is several orders of magnitude faster for the majority of benchmark inputs; for all other inputs our performance is, at worst, competitive with existing methods.

The structure of this paper is as follows. In Section 2, we provide a high-level overview of our algorithm, focusing on the (eager) construction of a graph-based representation of the search space (Section 2.2), followed by the (lazy) traversal of the search space (Section 2.3). We provide a worked example of the algorithm in Section 2.4, and an informal correctness argument in Section 2.5. Section 3 provides performance results, focusing on regular language difference (Section 3.1), regular intersection for large strings (Section 3.2), and bounded context-free intersection (Section 3.3). Section 4 briefly discusses related work, and we conclude in Section 5.

2. APPROACH

In the following subsections, we present our decision procedure for string constraints. Our goal is to provide expressiveness similar to that of existing tools such as DPRLE and Hampi [10, 12], while exhibiting significantly improved average-case performance. In Section 2.1, we formally define the string constraints of interest. Section 2.2 outlines our high-level graph representation of problem instances. We then provide an algorithm for finding satisfying assignments in Section 2.3, and give a brief illustrative example.

2.1 Definitions

In this work, we focus on a set of string constraints similar to those presented by Kieżun *et al.* [12], but without required a priori bounds on string variable length. In earlier work [10], we demonstrate that this type of string constraint can model a variety of common programming language constructs.

The set of well-formed string constraints is defined by the grammar in Figure 1. A constraint system S is a set of constraints of the form $S = \{C_1, \ldots, C_n\}$, where each $C_i \in S$ is derivable from *Constraint* in Figure 1. *Var* denotes a finite set of string variables $\{v_1, \ldots, v_m\}$. *ConstVal* denotes the set of string literals. For example, $v \in \texttt{ab}$ denotes that variable v must have the constant value `ab` for any satisfying assignment. We describe inclusion and non-inclusion constraints symmetrically when possible, using $\diamond$ to represent either relation (i.e., $\diamond \in \{\in, \notin\}$).

For a given constraint system S over variables $\{v_1, \ldots, v_m\}$, we write $A = [v_1 \leftarrow x_1, \ldots, v_m \leftarrow x_m]$ for the *assignment* that maps values $x_1, \ldots, x_m$ to variables $v_1, \ldots, v_m$, respectively. We define $[\![v_i]\!]_A$ to be the value of v_i under assignment A; for a *StringExpr* E, $[\![E \circ v_i]\!]_A = [\![E]\!]_A \circ [\![v_i]\!]_A$. For a *RegExpr* R, $[\![R]\!]$ denotes the set of strings in the language $L(R)$, following the usual interpretation of regular expressions. When convenient, we equate a regular expression literal like $\texttt{ab}^\star$ with its language. We refer to the negation of a language using a bar (e.g., $\overline{\texttt{ab}^\star} = \{w \mid w \notin \texttt{ab}^\star\}$).

An assignment A for a system S over variables $\{v_1, \ldots, v_m\}$ is *satisfying* iff for each constraint $C_i = E \diamond R$ in the system S, it holds that $[\![E]\!]_A \diamond [\![R]\!]$. We call constraint system S *satisfiable* if there exists at least one satisfying assignment; alternatively we will refer to such a system as a *yes–instance*. A system for which no satisfying assignment exists is *unsatisfiable* and a *no–instance*. A *decision procedure* for string constraints is an algorithm that, given a constraint system S, returns a satisfying assignment for S iff one exists, or "Unsatisfiable" iff no satisfying assignment exists.

2.2 Follow Graph Construction

We now turn to the problem of efficiently finding satisfying assignments for string constraint systems. We break this problem into two parts. First, in this subsection, we develop a method for eagerly constructing a high-level description of the search space. Then, in Section 2.3, we describe a lazy algorithm that uses this high-level description to search the space of satisfying assignments.

For a given constraint system I, we define a *follow graph*, G, as follows:

- For each string variable v_i, the graph has a single corresponding vertex $\texttt{node}(v_i)$.

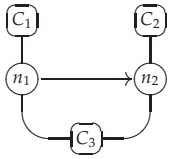
- For each occurrence of $\ldots v_i \circ v_j \ldots$ in a constraint in I, the graph has a directed edge from $\mathsf{node}(v_i)$ to $\mathsf{node}(v_j)$. This edge encodes the fact that the satisfying assignment for v_j must immediately follow v_i's.

We also maintain a mapping M from individual constraints in I to their corresponding path through the follow graph. For each constraint $C_h = v_j \diamond R$, we map C_h to path $[\mathsf{node}(v_j)]$. For each constraint C_i of the form $v_k \circ \ldots v_m \diamond R$, we map C_i to path $[\mathsf{node}(v_k), \ldots, \mathsf{node}(v_m)]$.

Figure 2 provides high-level pseudocode for constructing the follow graph for a given system. The follow_graph procedure takes a constraint system I and outputs a pair (G, M), where G is the follow graph corresponding to I, and M is the associated mapping from constraints in I to paths through G. For each constraint in I (line 4), we add edges for each adjacent pair of variables in the constraint (lines 5–7), and update M with the resulting path (line 8). For line 5, we assume that singleton constraints of the form $v_1 \diamond R$ are matched as well; this results in zero edges added (lines 6–7) and a singleton path $[\mathsf{node}(v_1)]$ (line 8).

As an example, consider the following constraint system and its associated follow graph:

$$C_1 = (v_1 \in \mathtt{a}^\star)$$
$$C_2 = (v_2 \in \mathtt{ab})$$
$$C_3 = (v_1 \circ v_2 \in \mathtt{ab})$$



We represent the graph G with circular vertices. The other vertices represent the domain of the mapping M. We assume $n_i = \mathsf{node}(v_i)$. The first two constraints result in the mapping from C_1 to $[n_1]$ and C_2 to $[n_2]$; the third constraint adds the mapping from C_3 to $[n_1, n_2]$. When convenient, we will use variables in place of their corresponding follow graph nodes.

2.3 Lazy State Space Exploration

Given a follow graph G, and a constraint-to-path mapping M, our goal is to determine whether the associated constraint system has a satisfying assignment. We treat this as a search problem; the search space consists of possible mappings from variables to paths through finite automata (NFAs). We find this variables-to-NFA-paths mapping through a backtracking depth-first search. If the search is successful, then we extract a satisfying assignment from the search result. If we fail to find a mapping, then it is guaranteed not to exist, and we return "Unsatisfiable." In the remainder of this subsection, we will discuss the search algorithm; we walk through a run of the algorithm in Section 2.4.

The NFAs used throughout the algorithm are generated directly from the regular expressions in the original constraint system; our implementation uses an existing algorithm due to Ilie *et al.* [11]. For constraints of the form $\ldots \in R$, we construct an NFA that corresponds to $L(R)$ directly. For constraints of the form $\ldots \notin R$, we eagerly construct an NFA that accepts $L(R)$. We then use a lazy version of the powerset construction to determinize and negate that NFA (e.g., [25]). We assume, without loss of generality, that each NFA has a single final state.

```
 1: datatype result =
 2:                | Unsat of result
 3:                | Sat of assignment → result
 4: datatype status =
 5:                | Unknown of status
 6:                | StartsAt of nfastate → status
 7:                | Path of nfapath → status
 8: datatype pos =
 9:              (constraint × int)
10: datatype searchstate =
11:             { next : variable;
12:               states : variable → pos → status}
13: datatype stepresult =
14:                | Next of searchstate → stepresult
15:                | Back of stepresult
16:                | Done of stepresult
17:
18: search(followgraph G, mapping M) =
19:   let Q : variable → pos → status = start_states(M)
20:   let O : searchstate = { next = nil; states = Q}
21:   let S : searchstate stack = [O]
22:   while S is not empty do
23:     let O_cur : searchstate = top(S)
24:     let R : stepresult = visit_state(O_cur, G, M)
25:     match R with
26:       | Next(O') → push(O', S)
27:       | Back → pop(S)
28:       | Done → return Sat(extract(O_cur))
29:   end while
30:   return Unsat
31:
32: visit_state(searchstate O, followgraph G, mapping M) =
33:   if ∀v : node ∈ G, all_paths(O.states[v]) then
34:     return Done
35:   if O.next = nil then
36:     O.next ← pick_advance(O, G, M)
37:   let (success, paths) = advance(O, G, M)
38:   if ¬success then
39:     return Back
40:   let O' : searchstate = copy(O)
41:   O'.next ← nil
42:   O'.states[O.next] ← paths
43:   foreach n : variable ∈ succ(O.next, G) do
44:     foreach p = (C, i) : pos s.t.
45:           O'.states[O.next][p] = Path(x) ∧
46:           O'.states[n][(C, i + 1)] = Unknown do
47:       O'.states[n][(C, i + 1)] ← StartsAt(last(x))
48:   return Next(O')
```

Figure 3: Lazy backtracking search algorithm for string constraints. The search **procedure performs an explicit search for satisfying assignments. Each occurrence of a variable in the constraint system is initially unconstrained (**Unknown**) or constrained to an NFA start state (**StartsAt**). Each call to** visit_state **attempts to move one or more occurrences from** Unknown **to** StartsAt **or from** StartsAt **to** Path**. The goal is to reach a** searchstate **in which each occurrence is constrained to a concrete** Path **through an NFA. Other procedures (e.g.,** start_states**,** extract**, and** advance**) are described in the text.**

2.3.1 The Search Algorithm

For clarity, we will distinguish between *restrictions* on variables imposed by the algorithm and *constraints* in the input constraint system. Our search starts by considering all variables to be unrestricted. We then iteratively pick one of the variables to restrict; doing this typically imposes further restrictions on other variables as well. The order in which we apply restrictions to variables does not affect the eventual outcome of the algorithm (i.e., "Satisfiable" or "Unsatisfiable"), but it may affect how quickly we find the answer. During the search, if we find that we have over-restricted one of the variables, then we backtrack and attempt a different way to satisfy the same restrictions. At the end of the search, there are two possible scenarios:

- At the end of a successful search, each occurrence of a variable in the original constraint system will be mapped to an NFA path; all paths for a distinct variable will have at least one string in common. We return "Satisfiable" and provide one string for each variable.
- At the end of an unsuccessful search, we have searched all possible NFA path assignments for at least one variable, finding no internally consistent mapping for at least one of those variables. There is no need to explore the rest of the state space, since adding constraints cannot create new solutions. We return "Unsatisfiable."

Figure 3 provides high-level pseudocode for the search algorithm. The main entry point is search (lines 18–30), which returns a result (line 1–3). An assignment (line 3) is a satisfying assignment that maps each variable to a string. The search procedure performs a depth-first traversal of a (lazily constructed) search tree; the stack S (line 21) always holds the current path through the tree. Each vertex in the search tree represents a mapping from string variables to restrictions; each edge represents the application of one or more additional restrictions relative to the source vertex.

Each iteration of the main loop (lines 22–29) consists of a call to visit_state. The visit_state procedure takes the current search state, attempts to advance the search, and returns a stepresult (lines 13–16) signaling success or failure. If visit_state returns Next, then we advance the search by pushing the provided search state onto the stack (line 26). If visit_state returns Back, then we backtrack a single step by popping the current state from the stack (line 27). If visit_state returns Done, then we extract a satisfying string assignment from the paths in current search state (line 28). Finally, if the algorithm is forced to backtrack beyond the initial search state, we return Unsat (line 30).

2.3.2 Manipulating the Search State

The searchstate type (lines 10–12) captures the bookkeeping needed to perform the search. The *next* element stores which string variable the algorithm will try to further restrict; once set, this will remain the same for potential subsequent visits to the same search state. The *states* element holds the restrictions for each variable for each occurrence of that variable in the constraint system. For example, in the constraint system

$$C_1 = (v_1 \circ v_1 \in R_1)$$

variable v_1 occurs at *positions* (lines 8–9) $(C_1, 1)$ and $(C_1, 2)$. The searchstate maps each variable at each position to a status (lines 4–7), which represents the current restrictions on that occurrence as follows:

1. Unknown (line 5) — This status indicates that we do not know where the NFA path for this variable occurrence should start. In the example, the $(C_1, 2)$ occurrence of v_1 will initially map to Unknown, since its start state depends on the final state of the v_1 occurrence at $(C_1, 1)$.
2. StartsAt (line 6) — This status indicates that we know at which NFA state we should start looking for an NFA path for this variable occurrence. In the example, the $(C_1, 1)$ occurrence of v_1 will initially map to StartsAt($\mathsf{nfa}(C_1).s$), where $\mathsf{nfa}(C_1).s$ denotes the start state of the NFA for regular expression R_1.
3. Path (line 7) — This status indicates that we have restricted the occurrence to a specific path through the NFA for the associated constraint. If a variable has multiple occurrences mapped to Path status, then those paths must agree (i.e., have at least one string in common).

Note that these restrictions are increasingly specific. Each non-backtracking step of the algorithm moves at least one variable occurrence from Unknown to StartsAt or from StartsAt to Path. Conversely, each backtracking step consists of at least one move in the direction Path → StartsAt → Unknown.

The majority of the pseudocode in Figure 3 deals with the manipulation of searchstate instances. The start_states call (line 19) generates the initial restrictions that start the search; it is defined for each variable v for each valid position (C, i) as follows:

$$\mathsf{start\_states}(M)[v][(C,i)] = \begin{cases} \mathsf{Unknown} & \textbf{if } i > 1 \\ \mathsf{StartsAt}(\mathsf{nfa}(C).s) & \textbf{if } i = 1 \end{cases}$$

The visit_state procedure advances the search by generating new search states (children in the search tree) based on a given search state (the parent). On lines 33–34, we check to see if all variable occurrences have a Path restriction. The corresponding NFA paths are required to agree by construction. In other words, the algorithm would never reach a search state with all Path restrictions unless the path assignments were internally consistent. We continue if there exists at least one non-Path restriction.

The call to pick_advance determines which variable we will try to restrict in this visit and any subsequent visits to this search state. This function determines the order in which we restrict the variables in the constraint system. The order is irrelevant for correctness as long as pick_advance selects each variable frequently enough to guarantee termination of the search. However, for non-cyclic parts of the follow graph, it is generally beneficial to select predecessor nodes (variables) in the follow graph before their successors. This is the case because visiting the predecessor can potentially change some of the successor's Unknown restrictions to StartsAt restrictions. We leave a more detailed analysis of search heuristics for future work.

The remainder of visit_state deals with tightening restrictions:

- The call to advance (line 37) performs lazy NFA intersection on all of the occurrences of variable $O.next$

to convert StartsAt restrictions to Path restrictions (or rule out that a valid path restriction exists, given the initial restrictions).

- If the call to advance succeeds, then the search state generation code of lines 42–47 uses the additional Path restrictions (if any) for $O.next$ to update $O.next$'s successors in the follow graph (if any; succ(v,G) returns the set of immediate successors of v in G). This step exclusively converts Unknown restrictions to StartsAt restrictions. The intuition here is that, if v_2 follows v_1 in some constraint, then the first state for that occurrence of v_2 must match the last state for v_1; $\mathsf{last}(x)$ (line 47) returns the last state in NFA path x.

Note that the first step (the call to advance) can potentially fail if $O.next$ proves to be over-restricted. When this occurs, we backtrack (lines 27 and 39) and return to a previous state, causing that state to be visited a second time. These subsequent visits will lead to repeated call to advance on the same parameters. We assume that advance keeps internal state to ensure that it exhaustively attempts all Path restrictions.

2.3.3 Finding NFA Paths Based on Restrictions

The advance function (called on line 37 of Figure 3) performs all automaton intersection operations during the search. Given some combination of Unknown, StartsAt, and Path restrictions on the occurrences of a given variable, the goal is to convert every StartsAt restriction to a Path restriction *while respecting all other restrictions*. This is necessary because the eventual goal is to find a single string assignment for each variable.

How we conduct the traversal for each variable depends on the restriction types for the variable's occurrences:

- An Unknown restriction indicates that, for the given occurrence, we do not know where the NFA path starts. However, we do know that (a) there should exist a valid (agreeing) path through this automaton starting at some start state; and (b) if the occurrence is the last position in a constraint, then the final state for the associated automaton must be reached.

  We use this information only if no other Path restrictions are present; this is necessary to ensure that we generate strings up to the correct maximum length (if we didn't involve all automata in the intersection, we could end up generating only strings that are short). Alternatively, we can avoid this step by imposing a length bound on variables.

- A StartsAt restriction requires a path through a given NFA starting at the given state; the path should agree with all other restrictions.

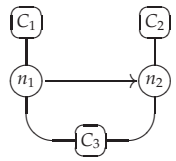
- A Path restriction requires that all other paths agree exactly with the current path.

We perform an explicit, lazy, search of the intersection (cross product) automaton. This is equivalent to a simultaneous depth-first traversal of the the various automata and paths; the traversal terminates if we simultaneously reach all desired final states. In addition, we must guarantee that, given the same searchstate, repeated calls to advance yield all possible non-repeating paths through the intersection automaton. We accomplish this by storing the search stack for NFA states between calls; if the stack is empty, we know we have exhausted all possible paths given the current constraints. Informally, the postcondition for advance is that any StartsAt restriction is replaced with a Path restriction, and any output Path restrictions agree on the concrete NFA path. If advance signals failure, then previous calls have exhausted all possible non-repeating paths through the intersection automaton.

2.4 Worked Example

Consider our previous example:

$$C_1 = (v_1 \in \mathtt{a}^\star)$$
$$C_2 = (v_2 \in \mathtt{ab})$$
$$C_3 = (v_1 \circ v_2 \in \mathtt{ab})$$



The initial searchstate (generated on lines 19–20 of Figure 3) would look as follows:

$$\begin{aligned}\{\ & next = \mathsf{nil};\\ & states = \{v_1 \mapsto \{(C_1,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_1).s);\\ &\qquad\qquad (C_3,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_3).s)\};\\ &\qquad v_2 \mapsto \{(C_2,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_2).s);\\ &\qquad\qquad (C_3,2) \mapsto \mathsf{Unknown}\ \ \}\}\}\end{aligned}$$

The main search procedure now visits this searchstate. The visit_state procedure, in turn, calls pick_advance (line 36). We assume $O.next$ is set to v_1, since it is first in a topological ordering of the follow graph. The advance procedure is called to intersect the language for C_1 with the prefixes for the language of C_3. Suppose the intersection (unluckily) results in a path matching a. This replaces the two StartsAt restrictions for v_1 with Path restrictions. On line 37, *paths* now equals:

$$\begin{aligned}\{\ & (C_1,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_1).s, \mathsf{nfa}(C_1).s]);\\ & (C_3,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_3).s, \mathsf{nfa}(C_3).q'])\ \ \}\end{aligned}$$

On lines 40–47, we create the next search state to visit. Because $v_2 \in \mathsf{succ}(v_1, G)$, and v_2 has an Unknown restriction on the correct occurrence, the final O' looks as follows:

$$\begin{aligned}\{\ & next = \mathsf{nil};\\ & states = \{v_1 \mapsto \{(C_1,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_1).s, \mathsf{nfa}(C_1).s]);\\ &\qquad\qquad (C_3,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_3).s, \mathsf{nfa}(C_3).q'])\ \ \};\\ &\qquad v_2 \mapsto \{(C_2,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_2).s);\\ &\qquad\qquad (C_3,2) \mapsto \mathsf{StartsAt}\ (\mathsf{nfa}(C_3).q')\ \ \}\}\}\end{aligned}$$

At this point visit_state returns (line 48) and O' is pushed onto the stack (line 26). On the next iteration, pick_advance selects v_2, since it is the only variable with work remaining. When we call advance, we notice a problem: C_2 requires that v_2 begin with a, but we have already consumed the a in C_3 using v_1. This means no NFA paths are feasible, and we return Back (line 39).

In search, we pop O_{cur} off the stack (line 27). On the next loop iteration, we revisit the initial search state. Since we previously set $O.next \leftarrow v_1$, we proceed immediately to the advance call without calling pick_advance. The advance procedure has only one path left to return: the trivial path that matches the empty string ϵ. At the end of visit_state, O' now equals:

$$\{\ next = \mathsf{nil};$$
$$states = \{v_1 \mapsto \{(C_1,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_1).s]);$$
$$(C_3,1) \mapsto \mathsf{Path}([\mathsf{nfa}(C_3).s])\ \};$$
$$v_2 \mapsto \{(C_2,1) \mapsto \mathsf{StartsAt}(\mathsf{nfa}(C_2).s);$$
$$(C_3,2) \mapsto \mathsf{StartsAt}\ (\mathsf{nfa}(C_3).s)\ \}\}\}$$

On the next iteration, pick_advance again selects v_2. A call to advance yields agreeing paths from $\mathsf{nfa}(C_2).s$ to $\mathsf{nfa}(C_2).f$ and from $\mathsf{nfa}(C_3).s$ to $\mathsf{nfa}(C_3).f$. On the final iteration, the all_paths check on line 33 is satisfied, and we extract the satisfying assignment from O_{cur} on line 28.

This example illustrates several key invariants: the algorithm starts exclusively with StartsAt and Unknown restrictions. Each forward step in the search tightens those restrictions by moving from StartsAt to Path and from Unknown to StartsAt. Any given search state is guaranteed to have mutually consistent restrictions. Once set, the only way to eliminate a restriction is by backtracking. Backtracking occurs only if, given the current restrictions, it is impossible to find an agreeing set of paths for the selected variable.

2.5 Correctness

Having described our algorithm, we now turn to an informal correctness argument. Decision procedures that return witnesses, in general, are required to be sound, complete, and terminate for all valid inputs. We discuss each of these aspects in turn, referring back to the definitions in Section 2.1 and the pseudocode of Figure 3 when necessary.

Definition *Soundness*:

$$\forall I, \mathsf{search}(\mathsf{follow\_graph}(I)) = \mathsf{Sat}(A) \Rightarrow$$
$$\forall (E \diamond R) \in I, [\![E]\!]_A \diamond [\![R]\!]$$

We assume the correctness of the follow_graph procedure. The start_states and visit_state procedures enforce the following invariants for NFA paths:

- The first variable occurrence in each constraint must have its path start with the start state for that constraint's NFA.
- All non-first variable occurrences in each constraint must have their paths start with the final state of their immediate predecessor in the constraint.
- The last variable occurrence in each constraint must have its path end with the final state for that constraint's NFA.

The first bullet is enforced by start_states (as defined in the text) using StartsAt restrictions; these restrictions are preserved when advance moves the StartsAt restrictions to Path restrictions. The second bullet is enforced directly by visit_state in lines 43–47 when moving Unknown restrictions to StartsAt restrictions.The third bullet is enforced by advance when generating paths.

Taken together, these conditions show exactly the right-hand side of the implication: for each constraint $C = (\ldots \diamond R)$, if we concatenate the variable assignments, we end up with a string w that must (by construction) take $\mathsf{nfa}(C).s$ to $\mathsf{nfa}(C).f$, showing $w \diamond R$.

Definition *Completeness*:

$$\forall I, \mathsf{satisfiable}(I) \Rightarrow \mathsf{search}(\mathsf{follow\_graph}(I)) \neq \mathsf{Unsat}$$

Intuitively, we want to show that for any satisfiable constraint system, there exists a path in a sufficiently-high search tree that reaches an "all paths" searchstate. This argument relies heavily on the completeness of advance, since that procedure essentially determines which child nodes we visit.

Definition *Termination*: search returns in a finite number of steps for all inputs.

A termination proof must show that the main loop on lines 22–29 of Figure 3 always exists in a finite number of steps. This follows from several facts:

- Each vertex in the search tree has a finite number of children, because advance generates a finite number of non-repeating paths through a cross-product NFA.
- For a given parent vertex in the search tree, we never visit the same child vertex twice. If we backtrack to the parent node, the advance is guaranteed to generate a distinct child node (or report failure).
- The tree has finite height because each step away from the root modifies at least one restriction in the direction of Path. Suppose we assume that all variable occurrences have Unknown restrictions except for one StartsAt restriction (the minimum), and also that we move only one restriction per step. In this case, the maximum height is $\Theta(2n)$ where n is the number of variable occurrences.

3. EXPERIMENTS

We present several experiments to evaluate the utility of our approach. We compare a prototype implementation in C++ with three publicly available tools: CFG Analyzer [1], DPRLE [10] and Hampi [12]. We also provide an indirect comparison with work by Veanes *et al.* [28, 27]. We evaluate several related tasks:

- **Set Difference.** In Section 3.1, we consider a benchmark used by Veanes *et al.* [27]. Given two regular expressions (a, b), the task is to compute a string in $L(a) \setminus L(b)$, if one exists. The benchmark consists of 10 regular expressions taken from real-world code [17]. We compare Hampi, DPRLE, and our prototype, running each on all 100 pairs of regular expressions.
- **Generating Long Strings.** In Section 3.2, we conduct an experiment used to evaluate the scalability of the Rex tool [28]. For each n between 1 and 1000 inclusive, the task is to compute a string in the intersection of `[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`. We compare Hampi, DPRLE, and our prototype.
- **Bounded Grammar Intersection.** In Section 3.3, we compare CFG Analyzer, Hampi, and our prototype on a grammar intersection task. We select 85 pairs of context-free grammars from a large data set [1]. The task, for each implementation, is to generate strings of length 5, 10, and 12, for each grammar pair.

	Regular Expression	Size
1.	`\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*([,;]\s*\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)*`	1.2 KB
2.	`\$?(\d{1,3},?(\d{3},?)*\d{3}(\.\d{0,2})?|\d{1,3}(\.\d{0,2})?|\.\d{1,2}?)`	399 B
3.	`([A-Z]{2}|[a-z]{2}[ ]\d{2}[ ][A-Z]{1,2}|[a-z]{1,2}[ ]\d{1,4})?([A-Z]{3}|[a-z]{3}[ ]\d{1,4})?`	425 B
4.	`[A-Za-z0-9](([ \.\-]?[a-zA-Z0-9]+)*)@([A-Za-z0-9]+)(([\.\-]?[a-zA-Z0-9]+)*)\.[ ]([A-Za-z][A-Za-z]+)`	390 B
5.	`(\w|-)+@((\w|-)+\.)+(\w|-)+`	442 B
6.	`[+-]?([0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)([eE][+-]?[0-9]+)?`	228 B
7.	`([\w\d.-]+)@{1}(([\w\d-]{1,67})|([\w\d-]+\.[\w\d-]{1,67}))\.((([a-z]|[A-Z]|\d){2,4})(\.([a-z]|[AZ]|\d){2})?)`	207 KB
8.	`(([A-Za-z0-9]+[_]+)|([A-Za-z0-9]+\-+)|([A-Za-z0-9]+\.+)|([A-Za-z0-9]+\++))*[A-Za-z0-9]+@((\w+\-+)|(\w+\.))*\w{1,63}\.[a-zA-Z]{2,6}`	65 KB
9.	`(([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.([a-zA-Z]{2,5}){1,25})+([;.](([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.([a-zA-Z]{2,5}){1,25})+)*`	369 KB
10.	`((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[,]{0,1}\s*)+`	1.3 KB

Figure 4: Regular expressions used for Experiment 1. The notation follows that of the .NET framework [27]; we use the EMCAscript interpretation of the expanded regular expression in the input format for Hampi and our prototype.

The experiments were conducted on a 2.8GHz Intel Core 2 Duo machine running Linux with 3.2GB addressable RAM. We use unmodified versions of Hampi (revision 24), DPRLE (revision 4), and CFG Analyzer (v. 2007-12-03), built from source using Sun Javac (v1.6.0_16) for Hampi and the OCaml native compiler (v3.10.2) for the others. We use the prebuilt binaries for STP [6] and MiniSAT [5] included in the Hampi distribution. We use ZChaff [21] (v.2007-03-12) as the underlying SAT solver for CFG Analyzer. Our prototype is written in C++ and built using the GNU C++ compiler (v4.3.3). We measure wall clock time unless otherwise specified. We run Hampi in server mode [12] to avoid the repeated cost of virtual machine startup; for the other three tools, the measured time includes process startup time for each data point.

We note that, at the time of writing, only DPRLE supports solving multivariable constraints and concatenation. We are aware of at least one effort to add support for multiple variables to Hampi, but at the time of writing, such an extension is not publicly available. In addition, We are not aware of any publicly available benchmarks that could provide a reasonable performance comparison while exercising features like concatenation. Consequently, we focus on established benchmarks that are compatible with the majority of available tools. We leave the analysis of search heuristics and other potential optimizations for future work.

3.1 Experiment 1: Regular Set Difference

In this experiment, we test the performance of DPRLE [10], Hampi [12], and our prototype on a set difference task. We reproduce an experiment used to test the symbolic difference construction by Veanes *et al.* [27]. This experiment uses ten benchmark regular expressions presented by Li *et al.* [17]; they are taken from real-world code. The task, for each pair of regular expressions (a, b), is to compute a string that occurs in $L(a)$ but not $L(b)$. This yields 100 distinct inputs for each tool: 90 yes-instances (whenever $a \neq b$) and 10 no-instances (when $a = b$).

The subject regular expressions are listed in Figure 4. Neither our prototype nor the other tools under consideration currently support repetition operations like `+`, `?`, and `{i,j}`, so we *expand* these operations into the equivalent combination of concatenations and disjunctions (e.g., `a?` becomes `or("", a)` in the input language for Hampi). These expressions are presented in the format used for the Microsoft .NET framework [22]. The *Size* column in Figure 4 shows the size of each regular expression after expansion. We note that there is a substantial range of sizes: from 228B (number two) to 369KB (number nine).

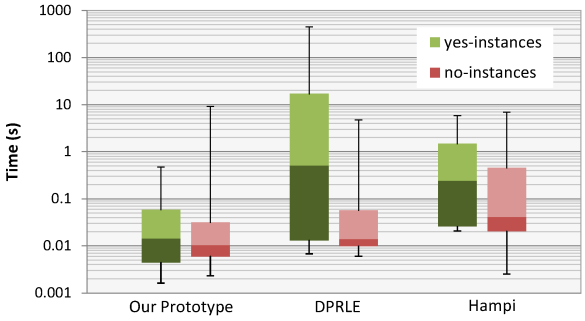


Figure 5: String generation time distributions, grouped by yes– and no–instances (left and right of each pair, respectively). The boxes represent the 25th through 75th percentile; the whiskers represent the 5th through 95th percentile.

We conducted the experiment as follows. For each pair of expanded regular expressions, we applied the appropriate transformations to create a valid constraint systems for each of the three tools. This required the following considerations (in both cases, giving any potential benefit to the other tools):

- Hampi requires a single fixed length bound for each input, and does not support searching for the empty string. For each pair of input regular expressions, we run Hampi on length bounds 1 through 10, inclusive. We terminate the search as soon as Hampi finds a string. In practice, we found that $k = 10$ allowed Hampi to correctly identify all yes-instances, while lower k did not.
- DPRLE requires automata descriptions for its input; it does not support regular expressions. Since our prototype performs a conversion from regular expressions to automata, we use that conversion algorithm to generate the DPRLE inputs. We do not count the conversion time towards DPRLE's running time; in practice we found that this made no significant difference.

Figure 5 summarizes the running times of the tools, grouped

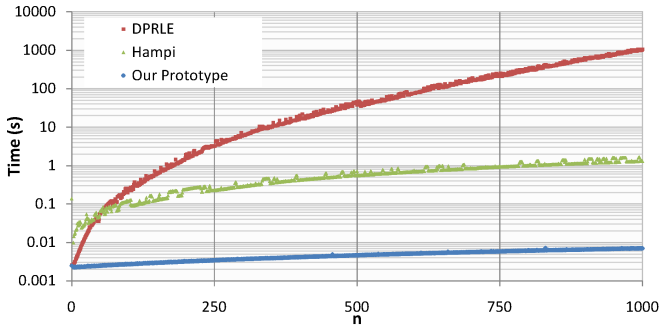


Figure 6: String generation times (log scale) for the intersection of the regular languages `[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`, for n between 1 and 1000 inclusive.

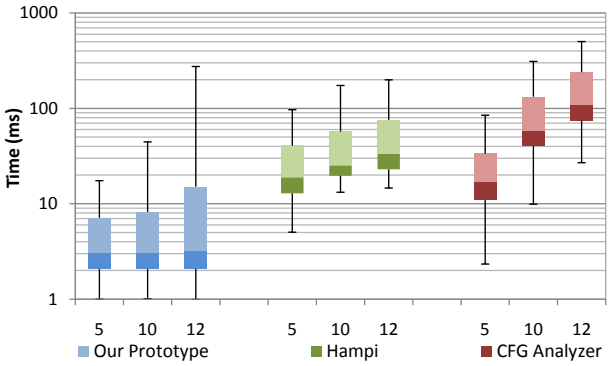


Figure 7: String generation times (log scale) for the intersection of context-free grammars. The grammar pairs were randomly selected from a dataset by Axelsson *et al.* [1]. Length bounds are 5, 10, and 12. Each column represents 85 data points; the bars show percentile 25 through 75 and the whiskers indicate percentile 5 through 95.

by yes-instances (90 datapoints per tool) and no instances (10 datapoints per tool). Note that the average time for our prototype tool on yes-instances is over an order of magnitude faster than DPRLE or Hampi, and that our tool exhibits relatively consistent timing behavior compared to the others. The performance gain arises from our construction of the state space corresponding to $L(\overline{b})$: this (potentially large) automaton is determinized and complementized lazily.

3.2 Experiment 2: Generating Long Strings

We hypothesize that our prototype implementation is particularly well-suited for underconstrained systems that require long strings. To test this hypothesis, we reproduce and extend an experiment used to evaluate the scaling behavior of Rex [28]. We compare the performance of Hampi, DPRLE, and our prototype. We also provide an indirect comparison with the results presented for Rex [28], which is not publicly available.

The task is as follows. For some length n, given the regular expressions

`[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`

find a string that is in both sets. For example, for $n = 2$, we need a string that matches both `[a-c]*a[a-c][a-c][a-c]` and `[a-c]*b[a-c][a-c]`; one correct answer string is `abcc`. Note that, for any n, the result string must have length $n + 2$. For Hampi, we specify this length bound explicitly; DPRLE and our prototype do not require a length bound.

For each n, we run the three tools, measuring the time it takes each tool to generate a single string that matches both regular expressions. Figure 6 shows our results. Our prototype is, on average, 118× faster than Hampi; the speedup ranges from 4.4× to 239×. DPRLE outperforms Hampi up to $n = 55$, but exhibits considerably poorer scaling behavior than both other tools. By comparison, the published Rex results [28] for $n = 1000$ show that tool taking approximately 140 seconds, or approximately 100× longer than Hampi, and 20,000× longer than our prototype on similar hardware. An informal review of the results shows that our prototype generates only a fraction of the NFA states; for $n = 1000$, DPRLE generates 1,004,011 states, while our prototype generates just 1,010 (or just 7 more than the length of the discovered path). These results suggest that lazy constraint solving can save large amounts of work relative to eager approaches.

3.3 Experiment 3: Length-Bounded Context-Free Intersection

In this experiment, we compare the performance of CFG Analyzer (CFGA) [1], Hampi [12], and our prototype. The experiment is similar in spirit to a previously published comparison between Hampi and CFGA: from a dataset of approximately 3000 context-free grammars published with CFGA, we randomly select pairs of grammars and have each tool search for a string in the intersection for several length bounds.

CFGA and Hampi differ substantially in how they solve this problem. Hampi internally generates a (potentially large) regular expression that represents all strings in the given grammar at the given bound. CFGA directly encodes the invariants of the CYK parsing algorithm into conjunctive normal form. For our prototype, we assume a bounding approach similar to that of Hampi. We use an off-the-shelf conversion tool, similar to that used by the Hampi implementation, to generate regular languages. We measure the running time of our tool by adding the conversion time and the solving time.

We randomly selected 200 pairs of grammars. Of these 200 pairs, 88 had at least one grammar at each length bound that produced at least one string. We excluded the other pairs, since they can be trivially ruled out without enumeration by a length bound check. We eliminated an additional three testcases because our conversion tool failed to produce valid output. We ran the three implementations on the remaining 85 grammar pairs at length bounds 5, 10, and 12, yielding 255 datapoints for each of the three tools. The ratio of yes–instances to no–instances was roughly equal. In terms of correctness, we found the outputs of Hampi and our prototype to be in exact agreement.

Figure 7 shows the running time distributions for each tool at each length bound. We note that our performance is, in general, just under an order of magnitude better than the other tools. In all cases, our running time was dominated by the regular enumeration step. We believe a better-integrated implementation of the bounding algorithm would significantly improve the performance for larger length bounds, thus potentially increasing our lead over the other tools.

4. RELATED WORK

In this section, we discuss closely related work, focusing on other string decision procedures and client applications.

There is significant theoretically-oriented work on word equations. Some of the problems discussed in this work are similar to those in the recent decision procedure literature, but focuses more on complexity bounds and decidability results. Kunc provides an overview of this area [14, 13]. The idea of treating a constraint solving problem as an explicit search problem is not new; many existing decision procedures are built around backtracking search (e.g., [24]).

The Hampi tool [12] is a solver for string constraints over fixed-size string variables. It supports regular languages, fixed-size context-free languages, and a number of operations (e.g., union, concatenation, Kleene star). Hampi has been extensively evaluated in static and dynamic analysis tools and for automatic test generation. Our new procedure supports similar operations to Hampi but without requiring fixed size bounds and with significant efficiency gains. In addition, our implementation supports multiple variables, while the currently available Hampi implementation does not.

The CFG Analyzer tool [1] is a solver for bounded versions of otherwise-undecidable context-free language problems. Problems such as inclusion, intersection, universality, equivalence and ambiguity are handled via a reduction to satisfiability for propositional logic in the bounded case. The Rex tool [27, 28] solves string constraints through a symbolic encoding of finite state automata into Z3 SMT solver [3]. An important benefit of this strategy is that string constraints can be readily integrated with other theories (e.g., linear arithmetic) handled by Z3. A disadvantage is that the encoding is relatively inefficient; in Section 3.1 and Section 3.2 we showed that Hampi and our prototype consistently outperformed Rex by up to four orders of magnitude.

The DPRLE tool [10] is a decision procedure for regular language constraints involving concatenation and subset operations. The tool focuses on generating entire sets of satisfying assignments rather than single strings: often constraints over multiple variables can yield multiple disjoint solution sets. The core algorithm of DPRLE has been formally proved correct in a constructive logic framework. Our new procedure supports similar operations to those allowed by DPRLE, but efficiently produces single witnesses rather than atomically generating entire solution sets. Nevertheless, our worst-case performance corresponds to that of DPRLE. For a large class of no–instances in which the contradiction occurs close to a right-most variable, our current algorithm necessarily generates a large subset of the NFA states that DPRLE generates by default.

A number of program analyses have been concerned with the values that string expressions can take on at run-time. Christensen *et al.* [2] check the validity of dynamically-generated XML. Similarly, Minamide [20] uses context-free grammars and finite state transducers to perform basic XHTML validity and cross-site scripting checks. Wassermann and Su build on Minamide's analysis to detect SQL injection vulnerabilities [29] and cross-site scripting vulnerabilities [30], by combining it with conservative static taint analysis.

Finally, there has been quite a bit of recent interest in automated test case generation. One goal of this line of work is to automatically produce an high-coverage test suite [16]. Path coverage is achieved by, in essence, computing the path predicates or guards associated with a large number of paths in the program and then treating them as constraints over the input variables. Solving the constraint system yields input variables that cause a given path to be taken. Early tools such as DART [8] or CUTE [18] focused largely on scalar constraints. More recent work has focused on the integration of string reasoning into such frameworks (e.g., [7, 19]).

5. CONCLUSION

Recent work on the analysis of string values has focused on providing external decision procedures for theories that model common programming idioms involving strings. Thus far, this work has focused on features such as support for concatenation operations [10], embedding into SMT solvers [27, 28], and bounded context-free languages [12].

In this paper, we present a constraint-solving algorithm for equations over string variables. Our algorithm has similar features to existing string decision procedures, but is designed to yield faster answers to yes-instances for large input constraint systems. We achieve this by treating the constraint solving problem as an explicit search problem. A key feature of our algorithm is that we instantiate the search space in an on-demand fashion.

We evaluated our algorithm by comparing our prototype implementation to publicly available tools like CFGA [1], DPRLE [10] and Hampi [12]. We used several sets of previously published benchmarks [12, 27]; the results show that our approach is up to four orders of magnitude faster than the other tools. We believe that as string constraint solvers continue to become more and more useful to other program transformations and analyses, scalability will be of paramount importance, and our algorithm is a step in that direction.

6. REFERENCES

[1] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental sat solver. In *International colloquium on Automata, Languages and Programming*, pages 410–422, 2008.

[2] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *International Symposium on Static Analysis*, pages 1–18, 2003.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[4] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[5] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[6] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer-Aided Verification*, pages 519–531, 2007.

[7] Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, June 9–11, 2008.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen.

DART: directed automated random testing. In *Programming Language Design and Implementation*, 2005.

[9] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*, 2008.

[10] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Programming Languages Design and Implementation*, pages 188–198, 2009.

[11] Lucian Ilie and Sheng Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.

[12] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *International symposium on Software testing and analysis*, pages 105–116, 2009.

[13] Michal Kunc. The power of commuting with finite sets of words. *Theory Comput. Syst.*, 40(4):521–551, 2007.

[14] Michal Kunc. What do we know about language equations? In *Developments in Language Theory*, pages 23–27, 2007.

[15] K. Lakhotia, P. McMinn, and M. Harman. Handling dynamic data structures in search based testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1759–1766, July 2008.

[16] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *Testing Academia and Industry Conference*, pages 95–104, September 2009.

[17] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Automated Software Engineering Short Paper*, November 2009.

[18] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *International Conference on Software Engineering*, pages 416–426, 2007.

[19] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering*, pages 134–143, 2007.

[20] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *International Conference on the World Wide Web*, pages 432–441, 2005.

[21] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, pages 530–535, 2001.

[22] MSDN. .net reference: Regular expression language elements. `http://msdn.microsoft.com/en-us/library/az24scfc(VS.71).aspx`. Technical report, 2001.

[23] George C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, 1997.

[24] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.

[25] Michael Sipser. *Introduction to the Theory of Computation*. Second edition. 1997.

[26] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Principles of Programming Languages*, pages 372–382, 2006.

[27] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. Solving extended regular constraints symbolically. Technical report, MSR, December 2009.

[28] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. Technical report, MSR, October 2009.

[29] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Programming Language Design and Implementation*, pages 32–41, 2007.

[30] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, 2008.

[31] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–374, 2009.

[32] Y. Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Usenix Security Symposium*, pages 179–192, July 2006.

[33] Yichen Xie and Alexander Aiken. Saturn: A SAT-based tool for bug detection. In *Computer Aided Verification*, pages 139–143, 2005.

[34] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.